Python Quick Start

Louis Sugy

Variables

- dynamic typing: `a = 3`, `b = "hello"`, `c = 4.2`
- base types: `int`, `float`, `bool` (`True` / `False`), `str`

Operators by descending order of precedence:

`**`	Exponentation
`~ + -`	Complement, sign
`* / // %`	Multiplication and division, Euclidean division, modulo
`+ -`	Addition and substraction
`>> <<`	Binary shift
`&`	Bitwise `and`
`^ |`	Bitwise `xor` and `or`
`<= >= < >`	Comparison
`== !=`	Equality
`= %= /= //= -= += *= **=`	Assignment
`is`, `is not`	Identity
`in`, `not in`	Membership
`and`, `or`, `not`	Logical operators

Conversions:

```
>>> a = "3.5"
>>> b = float(a)
>>> b
3.5
>>> c = int(b)
>>> c
3
```

Input / Output

- `a = input()` stores the next line in the standard input as a string in a variable. It takes an optional string parameter (a text to display before waiting for input)
- `print(a)` displays the representation of the variable on the standard output.

Conditions

```
if time < 9:
    print("Let's stay in bed a lil bit")
elif time == 9:
    print("Mmmm, time to wake up")
else:
    print("Oops I'm late")
```

Note: numbers are considered true if non-zero ; containers and strings are considered true if not empty.

Very important note: Python heavily relies on indentation. You should increase indentation by 4 spaces every time you enter a loop, condition, function, etc, and decrease indentation to mark the end of it.

Loops

These two pieces of code print 3, 5, 7, and 9:

```
for i in range(3, 11, 2):
    print(i)
```

```
i = 3
while i < 11:
    print(i)
    i += 2
```

`range([start,] end [,step])` generates integers from `start` (included) to `end` (excluded) with a step `step`. `step` can be negative and `start` greater than `end`. `start` defaults to 0 and `step` to 1.

You can exit a loop with the statement `break`, or skip to the next ieration with `continue`.

Strings

- defined with `a = "hello"` or `a = 'hello'` or `a = """hello"""` (multiline / docstring)
- non-mutable (you can get `a[3]` but not set it)
- *slices*: same as sequences, see in next part (e.g `"abcdef"[1:5:2]` is `"bd"`)
- Python 3 strings are Unicode by default (in Python 2 there was a separate type `unicode`)
- the backslash can be used to escape quotes or create special characters like end of line (`\n`) horizontal tab (`\t`), vertical tab (`\v`), etc. To prevent escaping, you can prefix a string with `r` (raw), e.g `r"a\nb"`

A few useful functions

`str.lower()`
`str.upper()`
	Returns a lower-case / uper-case copy of `str`

`str.strip([chars])`	Removes leading and trailing characters contained in the string `chars`. If not provided, defaults to whitespaces
`str.split(sep)`	Cuts the string into pieces according to the given separator and returns a list of these pieces
`str.join(iterable)`	Joins all the strings from the given sequence into a single string, with the separator `str`
`str.find(sub[, start[, end]])`	Returns the first position of the string `sub` in `str` between `start` and `end`, or −1 if not found

`str.startswith(sub)`
`str.endswith(sub)`
	Returns `True` if `str` starts / ends with `sub`, `False` otherwise.

Containers

The four main container types in Python are two **sequence** types (lists and tuples), one **associative** type (dictionaries), and a **collection** type (sets).

Lists

- by far the most used container type in Python
- dynamic arrays: can be modified, extended, truncated
- can contain elements of different types

It is easy to iterate on the elements of a list with a `for` loop. The following code displays `3`, `False`, `hello` and `4.2`:

```
mylist = [3, False, "hello"]
mylist += [4.2]
for element in mylist:
    print(element)
```

A few useful functions

`+ +=`	Concatenate lists
`len(list)`	Number of elements of a list
`list.clear()`	Makes the list empty
`list.append(x)`	Appends `x` at the end
`list.insert(i, x)`	Inserts `x` at position `i` (*)
`list.pop(i)`	Removes the element at position `i` and returns it (*)
`list.index(x [,start[,end]])`	Returns the position of the first occurence of `x` or raises an exception
`list.reverse()`	Reverses the list in place
`list.copy()`	Returns a copy of the list
`list.sort()`	See section *about sorting* below

(*) Warning: these operations are often slow (`O(n)`) because many elements need to be shifted.

Slices

- easy way to select a set of indices
- apply to strings, lists, tuples, etc

Before explaining slices: in Python negative indices count from the end, for example -1 *is the last element of a list, string, etc.*

Syntax: `[start]:[end][:step]`

- `start` is inclusive, `end` exclusive
- `start` is optional and defaults to 0
- `end` is optional and defaults to -1
- `step` is optional and defaults to 1

Here are some examples to see slices in action. Read them carefully and make sure you understand all of them:

```
>>> a = "hello world"
>>> a[::2]
'hlowrd'
>>> a[-1:0:-3]
'dooe'
>>> a[:]
'hello world'
```

```
>>> b = [1, 2, 3, 4, 5]
>>> b[1::2] = [7, 7]
>>> b
[1, 7, 3, 7, 5]
```

Tuples

- tuples hold a fixed number of values
- tuples can put together different types
- they are **immutable**: you cannot modify them, only give them a new value
- they are really useful combined with unpacking (cf next paragraph)

```
>>> a = (1, 2, 3, "Soleil")
>>> a[3]
'Soleil'
```

Trying to set `a[3]` would result in `TypeError: ’tuple’ object does not support item assignment`.

Tuples support for loops and slices like lists.

Unpacking

Unpacking is a very powerful feature of Python. It allows you to assign variables from an iterable (e.g a tuple, list, string, etc)) with the following rules:

- on the left of the `=` operator there must be as many variables as the size of the iterable, or at least one variable with a wildcard
- a wildcard variable, prefixed with `*`, becomes a list of as many items as the size of the iterable minus the number of other variables.
- this is recursive, meaning that if one of the element of the iterable is itself an iterable, you can also unpack its content (see examples below)

Here are some examples, read them carefully:

```
>>> a, *b, c = "Hello"
>>> print(a, b, c)
H ['e', 'l', 'l'] o
```

```
>>> a, (b, *c), *d = 3, "Hello", 4.2, ["World"]
>>> print(a, b, c, d)
3 H ['e', 'l', 'l', 'o'] [4.2, ["World"]]
```

```
>>> a, ((b, (c, d)), *e) = 1, ((2, (3, 4)), 5, 6)
>>> print(a, b, c, d, e)
1 2 3 4 [5, 6]
```

This feature can have use cases as simple as exchanging the values of two variables in one line without using a third variable:

```
>>> a = "queen"; b = "dancing"
>>> a, b = b, a
>>> print(a, b)
dancing queen
```

Dictionaries

- key–value associative structures
- optimized so that the size of the structure has no impact on the time required to get or set a value for a given key
- keys can be any immutable type such as a string, number, tuple, etc (more precisely, hashable types), and values can be anything

Example of basic use:

```
>>> dic = {"hello": "world", "lorem": "ipsum"}
>>> dic["hello"]
'world'
>>> dic["lorem"] = "hardy"
>>> dic
{'hello': 'world', 'lorem': 'hardy'}
```

You can easily check if a key is in the dictionary and iterate on keys:

```
>>> "hello" in dic
True
>>> "world" in dic
False
>>> for key in dic:
...     print(key, dic[key])
...
hello world
lorem hardy
```

A few useful functions

`len(dict)`	Number of key–value pairs
`dict.clear()`	Makes the dictionary empty
`dict.get(key [,default]`	Returns the value associated to `key`, if found, otherwise `default` if provided, or `None`
`dict.update(dict2)`	Adds all key–value pairs of `dict2` in `dict`, overwriting if necessary
`dict.keys()`	Returns a view (*) of `dict`'s keys
`dict.values()`	Returns a view (*) of `dict`'s values
`dict.items()`	Returns a view (*) of `dict`'s key–value pairs

(*) views are iterable objects that reflect the changes made in the source dictionary

Sets

- sets are unordered collections of unique items
- they are optimized to quickly add an item, check if an item belongs to the collection, remove an item
- they can contain elements from multiple types but only immutable types like strings, numbers, tuples, etc (more precisely, hashable types)

Example:

```
>>> s = set()
>>> s.add("hello")
>>> s.update({"lorem", "ipsum"})
>>> "hello" in s
True
>>> "world" in s
False
>>> "ipsum" not in s
False
```

```
>>> {1, 2} | {2, 3}
{1, 2, 3}
>>> {1, 2} - {2, 3}
{1}
>>> {1, 2} & {2, 3}
{2}
```

A few useful functions

`len(s)`	Number of elements in the set
`s.issubset(t)`	Tells wether `s` is a subset of `t`
`s.issuperset(t)`	Tells wether `s` is a superset of `t`
`s.union(t)`, `s | t`	Returns a set of all elements in `s` or `t`
`s.intersection(t)`, `s & t`	Returns a set of elements that are in both `s` and `t`
`s.difference(t)`, `s - t`	Returns a set of the elements of `s` which are not in `t`
`s.symmetric_difference(t)`, `s ^ t`	Returns a set of the elements that are either in `s` or `t` but not both
`s.copy()`	Returns a copy of `s`

Functions

- functions are defined with the `def` keyword, a name, arguments and a block of code
- arguments can be given default values and then be optional
- optional arguments can be defined by position in order from the left, or by their name (see examples below)
- the function returns a value, or packs multiple comma-separated return values in a tuple, with the `return` statement

Examples:

```
>>> def f(a, b=3, c=7):
...     return a + b + c
...
>>> f(0)
10
>>> f(0, 0)
7
>>> f(0, 0, 0)
0
>>> f(0, c=0)
3
```

Parameters unpacking

- functions args can be highly flexible with the use of wildcards
- `*args` will be a tuple of all unmatched positional arguments
- `**kwargs` will be a dictionary of all unmatched keyword arguments (see examples below)

```
>>> def f(*args, **kwargs):
...     print(args)
...     print(kwargs)
...
>>> f(1, 2, 3, soleil=True, lune=False)
(1, 2, 3)
{'soleil': True, 'lune': False}
```

Recursive functions

Functions can call themselves. For example, in order to calculate the factorial of an integer n (i.e the product $1 \times 2 \times \cdots \times n$), you could write:

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

Note: Python sets a default depth limit of about 1000 calls, so recursive functions are expected to be used on cases where the depth is small, for example proportional to the logarithm of the size of a data structure

About Sorting

- `mylist.sort()` will sort the list in ascending order if its elements can be compared
- `sorted(myList)` returns a sorted copy of the list
- the parameter `reverse` allows to choose the order: descending if `True`, ascending if `False`. It defaults to `False`
- the parameter `key` allows to specify a function that takes an element of the list and returns the criteria that the algorithm should use to compare elements to each other (see example below)

```
>>> def last_name(person):
...     return person["last_name"]
...
>>> persons = [
...     {"first_name": "Linus",
...      "last_name": "Torvalds"},
...     {"first_name": "Guido",
...      "last_name": "Van Rossum"},
...     {"first_name": "Richard",
...      "last_name": "Stallman"}
... ]
>>> sorted(persons, key=last_name)
[{'first_name': 'Richard',
  'last_name': 'Stallman'},
 {'first_name': 'Linus',
  'last_name': 'Torvalds'},
 {'first_name': 'Guido',
  'last_name': 'Van Rossum'}]
>>> sorted(persons, key=last_name, reverse=True)
[{'first_name': 'Guido',
 'last_name': 'Van Rossum'},
 {'first_name': 'Linus',
  'last_name': 'Torvalds'},
 {'first_name': 'Richard',
  'last_name': 'Stallman'}]
```

Comprehensions

- another powerful feature of Python
- define list or generator expression inline in a natural / mathematical way

Syntax: `... (for ... in ...)* if ...`

Examples:

```
>>> [i**2 for i in range(10) if i % 3 == 1]
[1, 16, 49]
```

```
>>> [j for i in range(5) for j in range(i)]
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

```
>>> sum(3*i for i in range(42))
2583
```

Going Further

- Modules: `https://www.tutorialspoint.com/python/python_modules.htm`
- Object Oriented Programming: `https://realpython.com/python3-object-oriented-programming/`
- Generators: `https://wiki.python.org/moin/Generators`
- Regex: `https://docs.python.org/3.7/library/re.html`

Python docs: `https://docs.python.org/3/contents.html`